PepsiCo: A collaborative filtering Recommender System based on Non-negative Matrix Factorization.

Ignacio Carrasco Ortega

A thesis submitted in conformity with the requirements
for the MSc in Economics, Finance and Computer Science

University of Huelva & International University of Andalusia

Julio, 2020

PepsiCo: A collaborative filtering Recommender System based on Non-negative Matrix Factorization.

Ignacio Carrasco Ortega **ignaciodeloyola.carrasco028@alu.uhu.es**

Antonio Javier Tallón Ballesteros **antonio.tallon@diesia.uhu.es**

PhD Lecturer. Department of Electronic, Computer Systems and Automation Engineering. University of Huelva.

MSc in Economics, Finance and Computer Science

University of Huelva & International University of Andalusia

Abstract

This project tries to solve a problem that the PepsiCo company proposes to us, which needs to improve the sales performance of a chain of supermarkets and hypermarkets in a member country of the European Union.

To solve the problem, a recommender system based on non-negative matrix factorization is proposed, which is a type of collaborative filtering of the model-based type. This model will make recommendations to the lowest performing stores in this chain to transform them into higher performing stores.

To optimize the model, a 5-fold cross-validation process is carried out, in which the value of the latent factors (k) is optimized, minimizing the error metrics RMSE and MAE for each pair of k choosing the value of k that least RMSE and MAE produces.

To implement the model, a specific recommender systems library, a framework called Surprise, will be used in the python programming language.

Key words: collaborative filtering (CF), non-negative matrix factorization (NMF), recommender system (RecSys).

Acknowledgments

I am extremely grateful to Pedro Cadahia for guiding me throughout this project, as well as for his advice and help.

I thank Antonio Tallón for his comments and guidance in contrasting the papers on which this work is based.

Table of Contents

List of Tables

List of Figures

1. Problem to solve

PepsiCo Company has the 2019 monthly sales data of a well-known supermarket and hypermarket chain belonging to a member country of the European Union. Using these data, they prepared a classification of each of the chain's stores, according to their sales (of PepsiCo products) and their market share, as shown in the following figure:

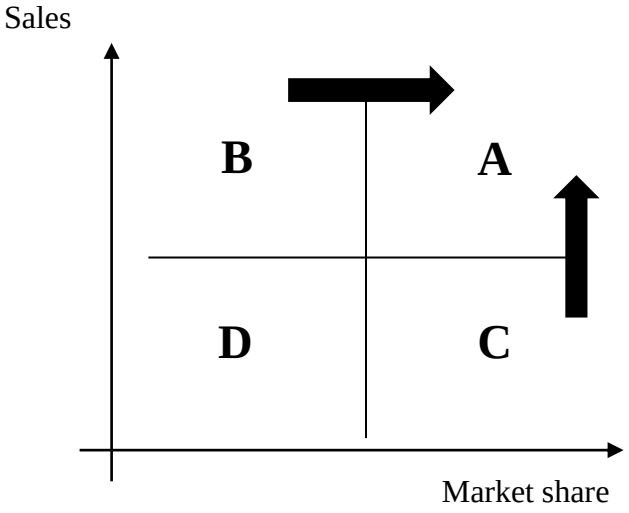

Figure 1. Store classification scheme. Source: Own elaboration.

The Project that PepsiCo proposes to us, is through the classification shown in the previous figure, to make type B stores and type C stores go to type A through a recommender system, which recommends to type B and type C stores the products that are sold in type A stores and are getting better performance.

This master's thesis will focus on the development of this **recommender system**.

2. The Dataset

The Dataset that PepsiCo makes available to us is a dataset as follows:

- 727666 instances.
- 23 variables

In which the monthly sales, market share, and units sold during 2019 of 496 stores of the previously mentioned chain are shown. Other variables such as the description of the products are also shown (in this work the description of the products will be coded), the class of the products, the barcode and the type of store among other variables, as shown in the following figure:

```
['BEGIN_MONTH_KEY', 'END_MONTH_KEY', 'COUNTRY_KEY', 'COUNTRY_DESC',
 'CHAIN_TYPE_KEY', 'CHAIN_TYPE_DESC', 'STORE_KEY', 'STORE_DESC',
 'CODE_POSTAL', 'CITY_DESC', 'REGION_DESC', 'CWT_GROUP_CLASS_KEY',
 'CWT_GROUP_CLASS_DESC', 'CWT_CLASS_KEY', 'CWT_CLASS_DESC', 'BARCODE',
 'ITEM_DESC', 'SALES_WO_VAT_ML', 'EVOL_SALES_WO_VAT_ML',
 'SALES_MARKET_SHARE_STORE', 'EVOL_SALES_MARKET_SHARE_STORE',
 'UNIT_SALES', 'EVOL_UNIT_SALES'],
```

Figure 2. fraction of available dataset.

To better work with the dataset, we grouped it by store and by product, and added only the quantitative variables, averaging sales, market share, and units sold. In this way, we annualize these numerical variables to have only one data instead of 12 (corresponding to the months of 2019). This grouped dataset would have the form:

- 74153 instances
- 5 variables

	STORE_KEY	ITEM_DESC	SALES_WO_VAT_ML	SALES_MARKET_SHARE_STORE	UNIT_SALES
0	1701	A	9.992500	0.005788	4.000000
1	1701	B	12.185333	0.001495	3.533333
2	1701	C	8.082381	0.001039	4.000000
3	1701	D	29.785625	0.003830	13.937500
4	1701	E	8.116667	0.001208	4.166667

Table 1. First 5 instances of the grouped dataset.

Once this is done, we eliminate the products whose sales are negative during 2019 (corresponding to returns from other stores) and the products whose sales are 0 during 2019 so as not to incur calculation errors as we will see later.

We check if this grouped dataset has missing values:

```
STORE_KEY                0
ITEM_DESC                0
SALES_WO_VAT_ML          0
SALES_MARKET_SHARE_STORE 0
UNIT_SALES               0
```

Figure 3. Missing values per variable.

As this grouped dataset does not contain missing values, there is no need to do any additional preprocessing.

Since the recommender system that we will use requires a user-item matrix, in which the users would be the stores, the items would be the products and the values would be the sales that each store makes of a certain product, we created a dataset in which the instances correspond to each of the 496 stores, the columns correspond to the products, and the interior values correspond to the sales that each store makes of each product.

Because not all stores sell all products, the result will be a sparse matrix, in which we will only have a record of the sales that a store makes of a certain product, with 0 being the products that the store does not sell. These 0 values are what we have to infer to make recommendations.

This last dataset that we have created has the form of a user-item matrix with which matrix factorization-based recommender systems usually work. In our case, as we will see later, the Surprise library that we will use to implement our model, generates this user-item matrix internally, so we will have to input 3 columns from the grouped sales matrix calculated in the previous paragraph: store, item and sales, so that the framework Surprise generates the user-item matrix.

The user-item matrix that most matrix factorization-based recommender systems would work with has the form:

- 496 instances (number of stores)
- 275 variables (number of products)

ITEM_DESC	A	B	C	D	E	F	G	H	I	J	L
STORE_KEY											
1701	9.992500	12.185333	8.082381	29.785625	0.000	8.116667	0.0	0.00	0.0	8.785000	20.262857
1703	6.234545	7.833333	3.035263	3.467500	4.134	3.187273	0.0	4.98	0.0	7.018000	4.221429
1705	14.185385	4.587500	6.233889	4.783000	0.000	6.956000	0.0	0.00	0.0	14.238333	8.997500
1707	16.470000	8.042667	0.000000	9.760000	0.000	0.000000	0.0	0.00	0.0	21.332500	6.002222
1709	12.631429	13.194000	7.940435	13.037000	0.000	6.910000	0.0	0.00	0.0	40.702500	13.838750

Table 2. First 5 instances of the sparse matrix user-item.

The missing values proportion of user-item matrix is shown in the next figure:

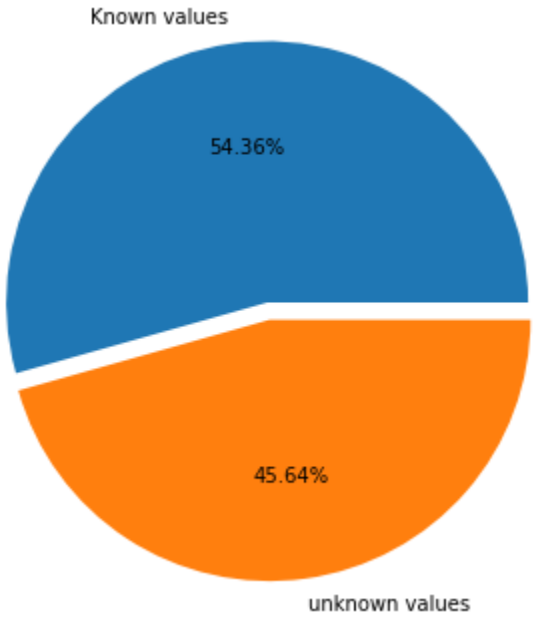

Figure 4. Pie chart of known and unknown values.

3. Introduction

The objective of a Recommender system (RecSys) is to recommend relevant items for users, based on their preference. In our case, the objective is to recommend relevant products for stores based on the underlying features behind the store`s sales. These features will be latent factors that our model will infer how we will explain later.

Along with this project, the store to which the recommendation is provided is referred to as the user, the product being recommended is referred to as an item and the values of the matrix user-item denoted as R, represent the sales.

The main families of methods for Recommender systems are content based systems, collaborative filtering systems, and hybrid systems (which use a combination of the other two):

- **Content-based recommender system:** This method leverages the features of items to recommend other similar items. For example, if I am browsing for solid colored t-shirts on Amazon, a content based recommender might recommend me other t-shirts or solid colored sweatshirts because they have similar features (sleeves, single color, shirt, etc).
- **Collaborative filtering based recommender systems:** This method uses the actions of users to recommend other items. The underlying assumption of the collaborative filtering approach is that if person A has the same opinion as a person B on a set of items, A is more likely to have B's opinion for a given item than that of a randomly chosen person. Collaborative Filtering (CF) has two main implementation strategies:
    - **Memory-based:** This approach uses the memory of previous users interactions to compute users similarities based on items they have interacted with (user-based approach) or compute items similarities based on the users that have interacted with them (item-based approach). User-based collaborating filtering uses the patterns of users similar to me to recommend a product (users like me also looked at these other items). Item-based collaborative filtering uses the patterns of users who browsed the same item as me to recommend me a product (users who looked at my item also looked at these other items). A typical example of this approach is User Neighbourhood-based CF, in which the top-N similar users (usually computed using Pearson correlation) for a user are selected and used to recommend items those similar users liked, but the current user have not interacted yet.
    - **Model-based:** This approach, models are developed using different machine learning algorithms to recommend items to users. There are many model-based CF algorithms, like neural networks, bayesian networks, clustering models, and latent factor models such as Singular Value Decomposition and, probabilistic latent semantic analysis.

- **Hybrid methods:** Recent research has demonstrated that a hybrid approach, combining collaborative filtering and content-based filtering could be more effective than pure approaches in some cases. These methods can also be used to overcome some of the common problems in recommender systems such as **cold start** and **the sparsity problem**.

To resolve our Recommender system, we will use a collaborative filtering, model-based approach: a variation of the latent factor model Singular Value Decomposition (SVD).

3.1. Latent factor based collaborative filtering

The basic idea of collaborative filtering methods is that these missing values can be imputed because the observed values are often highly correlated across various users and items. This similarity can be used to make inferences about missing values. Most of the collaborative filtering methods focus on leveraging either inter-item correlations or inter-user correlations for the prediction process.

Latent factor based models can be considered as a direct method for matrix completion. It estimates the missing entries of the rating matrix R, to predict what items a user will most probably like other than the ones they have rated. The basic idea is to exploit the fact that significant portions of the rows and columns of the rating matrix are correlated. As a result, the data has built-in redundancies and the sales matrix R can be approximated by a low-rank matrix. The low-rank matrix provides a robust estimation of the missing entries.

The method of approximating a matrix by a low-rank matrix is called **matrix factorization**. There are many different ways to factor matrices, Singular value decomposition (SVD) is particularly useful for making recommendations. At a high level, SVD is an algorithm that decomposes a matrix R into the best lower rank (i.e. smaller/simpler) approximation of the original matrix R. Mathematically, it decomposes R into two unitary matrices and a diagonal matrix:

$$\mathbf{R=U\Sigma V^T}$$

where R is the user-item matrix where the values are the sales, U is the user "features" matrix, Σ is the diagonal matrix of singular values (essentially weights), and V^T is the item "features" matrix. U and V^T are orthogonal and represent different things. U represents how much users "like" each feature and V^T represents how relevant each feature is to each item.

To get the lower rank approximation, we take these matrices and keep only the top k features, which we think of as the k most important underlying taste and preference vectors.

The matrix factorization problem in latent factor based model can be also formulated as an optimization problem given by:

$$\textbf{minimize U, V} \quad \sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{rij} - (\mathbf{U}.\mathbf{VT})\mathbf{ij})\mathbf{2} \qquad (1)$$

where U and V are matrices of dimension m × k and n × k respectively, where k is the number of latent factors. However, in the above setting it is assumed that all the entries of the rating matrix R are known, which is not the case with sparse rating matrices. Fortunately, latent factor model can still find the matrices U and V even when the rating matrix R is sparse. It does it by modifying the cost function to take only known rating values into account. This modification is achieved by defining a weight matrix W in the following manner:

Wij = 1 if rij is known; 0 if rij is unknown

Then, we can reformulate the optimization problem as:

$$\textbf{minimize U, V} \quad \sum_{i=1}^{m}\sum_{j=1}^{n}\mathbf{Wij}(\mathbf{rij} - (\mathbf{U}.\mathbf{VT})\mathbf{ij})\mathbf{2} \qquad (2)$$

Since the rating matrix R is sparse, so the observed set of ratings is very small. As a result, it might cause over-fitting. A common approach to address this problem is to use regularization. The optimization problem with regularization is given by:

$$\textbf{minimize U, V} \quad \sum_{i=1}^{m}\sum_{j=1}^{n}\mathbf{Wij}(\mathbf{rij} - (\mathbf{U}.\mathbf{VT})\mathbf{ij})\mathbf{2} + \boldsymbol{\lambda}\mathbf{kUk2} + \boldsymbol{\lambda}\mathbf{kVk2} \qquad (3)$$

The regularization parameter λ is always non-negative and it controls the weight of the regularization term. There are many variations to the unconstrained matrix factorization formulation (equation 3) depending on the modification to the objective function and the

constraint set. In this project, due to the sales values cannot be negative, we will use the next variation, which will be explained in the next point:

- **Non-negative matrix factorization (NNMF)**

This variation is very similar to SVD, explained at a high level above. The difference is that U and V are kept positive.

4. The model

Non-negative matrix factorization may be used for value matrices that are or should be non-negative. As in our case, store sales cannot be negative, so it seems an appropriate model. The major advantage of this method is the high level of interpretability it provides in understanding the user-item interactions. The main difference from other forms of matrix factorization as SVD is that the latent factors U and V must be non-negative. Therefore, optimization formulation in non-negative matrix factorization is given by:

$$\textbf{minimize U, V} \qquad \sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{Wij}(\mathbf{rij} - (\mathbf{U}.\mathbf{VT})\mathbf{ij})\mathbf{2} + \boldsymbol{\lambda}\mathbf{kUk2} + \boldsymbol{\lambda}\mathbf{kVk2} \qquad (4)$$

subject to $\mathbf{U} \geq 0$, $\mathbf{V} \geq 0$

Our implementation follows that suggested in [1], which is equivalent to [2] in its non-regularized form. Both are direct applications of NMF for dense matrices [3].

To solve the optimization problem (equation 4) we will use stochastic gradient descent (SGD), one of the many optimization algorithms available.

The optimization procedure is a (regularized) stochastic gradient descent with a specific choice of step size that ensures non-negativity of factors, provided that their initial values are also positive.

After we have solved the optimization problem in equation 4 for U and V, then we can use them for predicting the values. The predicted values of user i for item j, denoted by r̂ij, is given by:

$$\hat{r}\boldsymbol{ij} = \sum_{s=1}^{k} \mathbf{uis} \cdot \mathbf{vjs} \qquad (5)$$

5. The code

Once we have covered the necessary background to implement the basics of matrix factorization, now we are ready to implement a NMF based collaborative filter to make recommendations.

We will use built-in functions in python for the implementation, specifically a framework called **Surprise.** Surprise is an easy-to-use Python scikit for recommender systems.

5.1. Loading data and preprocessing

First, we import the necessary tools and libraries from Python:

```
import pandas as pd # Data processing and analysis
import numpy as np # Linear algebra and scientific computing

from surprise import NMF # Non-negative matrix factorization
from surprise import Dataset # To Load dataframes to surprise
from surprise.model_selection import cross_validate # To evaluate the model
from surprise import Reader # to set the value's scale
```

Then, we load the dataset, do some replacing and cleaning, and set some options from the dataframe library to see the full documents:

```
df4 = pd.read_csv(r"E:\Product_sales.csv", sep=";", decimal=",") # To read the csv file
df4.columns=df4.columns.str.replace('{','') # To clean some special caracters
df4.columns=df4.columns.str.replace('}','') # To clean some special caracters

pd.set_option('display.max_columns',df4_sal.shape[1]); # To set max columns to display
pd.set_option('display.max_rows',df4.shape[0]); # To set max rows to display
pd.set_option('display.float_format', lambda x: '%.2f' % x) # To set two decimals
```

Then we group by store and product the dataset, annualize the quantitative variables:

```
# To group by store and product the dataset, and  annualize sales, market share and sold products
df4_sales = df4.groupby(['STORE_KEY','ITEM_DESC'], as_index=False).agg({'SALES_WO_VAT_ML':['mean'],
                                                                        'EVOL_SALES_WO_VAT_ML':['mean'],
                                                                        'SALES_MARKET_SHARE_STORE':['mean'],
                                                                        'EVOL_SALES_MARKET_SHARE_STORE':['mean'],
                                                                        'UNIT_SALES':['mean'],
                                                                        'EVOL_UNIT_SALES':['mean']})
```

and do some preprocessing:

```
df4_sales.columns = df4_sales.columns.droplevel(1)

indexsales0 = []
indexsalesneg = []

# To look for sales = 0
for i in df4_sales.SALES_WO_VAT_ML.index[df4_sales.SALES_WO_VAT_ML == 0]:
    indexsales0.append(i)

# To look for sales < 0
for j in df4_sales.SALES_WO_VAT_ML.index[df4_sales.SALES_WO_VAT_ML < 0]:
    indexsalesneg.append(j)

#drop products with SALES_WO_VAT_ML = 0.
df4_sales.drop(indexsales0,axis=0,inplace=True)
#drop products with SALES_WO_VAT_ML < 0.
df4_sales.drop(indexsalesneg,axis=0,inplace=True)
```

Then we configure the settings of Surprise library. Keep in mind that the surprise library configures the user-item matrix internally. For this reason, our input is not the user-item matrix directly, but we give it the user, item and sales columns and the library configure it internally:

```
# To set the values scale
reader = Reader(rating_scale=(df4_sales.SALES_WO_VAT_ML.min(), df4_sales.SALES_WO_VAT_ML.max())
# To load the dataframe in scale
data = Dataset.load_from_df(df4_sales[['STORE_KEY','ITEM_DESC','SALES_WO_VAT_ML']], reader)
```

And finally, we configure the pie chart of the proportion of known values of the user-item matrix using matplotlib:

```
# To plot the pie chart of known and unknown values
%matplotlib inline
from matplotlib import pyplot as plt

exp_vals = [(df4_sal.shape[0]*df4_sal.shape[1])-(df4_sal.isnull().sum().sum()),df4_sal.isnull().sum().sum()]
exp_labels = ['Known values','unknown values']

plt.axis('equal') # To set the axis equal
plt.pie(exp_vals,labels=exp_labels,radius=1.5, autopct = '%0.2f%%', explode = [0,0.1])
plt.show()
```

Once we have loaded all the necessary libraries, we have the dataset ready, and we have prepared all the requirements of the surprise library, we are ready to optimize the model.

5.2. Optimizing the model

An important decision is the number of latent factors to factor the user-item matrix. The higher the number of latent factors, the more precise is the factorization in the original user-item matrix reconstructions. Therefore, if the model is allowed to memorize too many details of the user-item matrix, it may not generalize well for data it was not trained on and would tend to overfitting. Reducing the number of factors increases the model generalization.

We are going to create a training and validation process and optimize **k** by minimizing Root Mean Square Error **RMSE** and Absolute Mean Error **MAE.**

Intuitively, the Root Mean Square Error will continuously decrease on the training set as k increases (because I am approximating the user-item matrix with a higher rank matrix). On the validation set, however, the error will eventually start increasing because the training set is an overfit representation of user sales.

To optimize k, we are going to test the model performance via 5-fold cross-validation. In 5-fold cross-validation, the dataset is partitioned into 5 equal-sized subsets. Of the 5 subsets, a single subset is retained as the validation data for testing the model, and the remaining 4 subsets are used to train the model. The cross-validation process is then repeated 5 times, with each of the 5-subsets used exactly once as the validation data.

In order to optimize the number of latent factors, k, we are going to repeat the process described in the previous paragraph a number of times equal to the smallest dimension of the user-item matrix used, in this case, 276. Every 2 latent factors, we will perform the cross-validation process and obtain the mean of the RMSE and MAE metrics. We will store these metrics every 2 k, and in this way, we will be able to graphically represent them and see for which k values are obtained lower RMSE and MAE.

The lines of code to implement what is explained above are as follows:

```
Avg_RMSE = [] # We create a empty list to store the mean RMSE
Avg_MAE = [] # We create a empty list to store the mean MAE

# Run 5-fold cross-validation and store the results
for k in range(2,df4_sal.shape[1],2):
    # We will use the NMF algorithm.
    algo = NMF(n_factors=k, reg_pu = 0.02, reg_qi= 0.02, biased= False, verbose=False)
    cv = cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=False)

    rmse_mean =  cv['test_rmse'].mean()
    mae_mean = cv['test_mae'].mean()

    Avg_RMSE.append(rmse_mean)
    Avg_MAE.append(mae_mean)
```

Once we have obtained the RMSE and MAE averages for each pair of k values, we represent them graphically. To graphically represent the RMSE first we load matplotlib:

```
%matplotlib inline
from matplotlib import pyplot as plt
```

After that, we represent k values and RMSE using the following code:

```
kValue = np.arange(2,df4_sal.shape[1],2) # To set a range to K.

plt.figure(figsize=(10,10)) # To set figure size

# To set the plot settings
width = 1/1.5
plt.plot(kValue, Avg_RMSE,'go--', linewidth=1, markersize=3)

plt.xlabel('k Values') # X axis label
plt.ylabel('RMSE per K') # Y axis label
plt.grid('on')

plt.show() # To show only the plot
print(min(Avg_RMSE))
```

The following figure is the one we get by executing the code above:

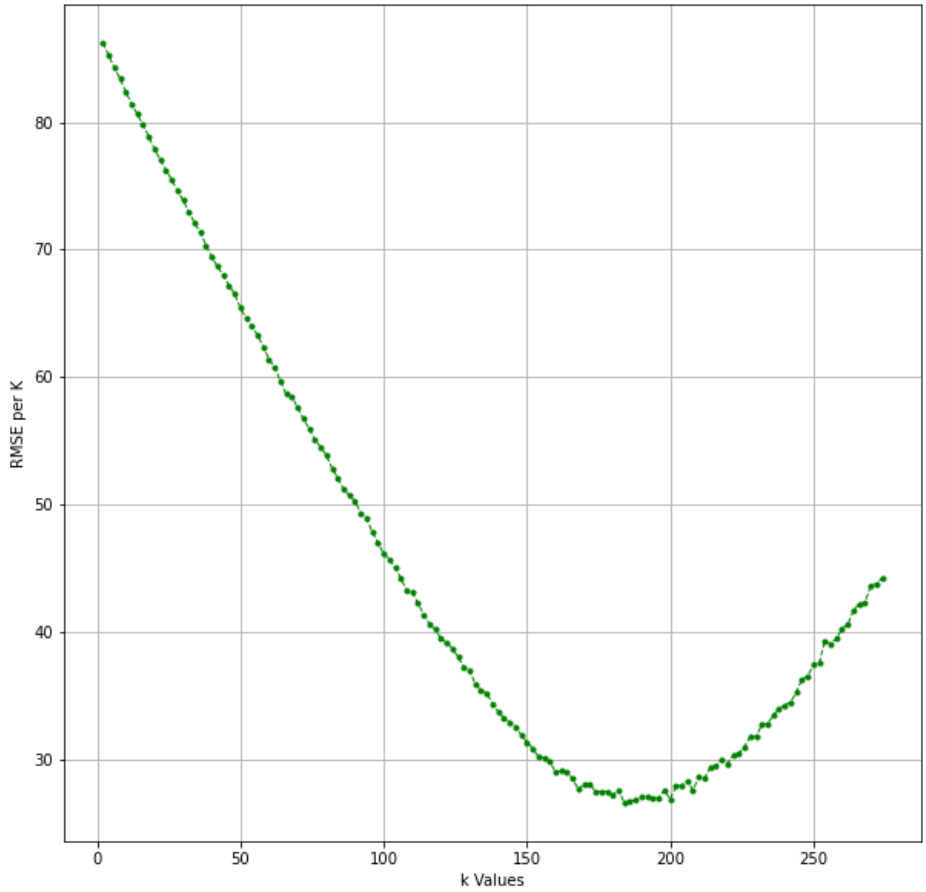

Figure 5. K values and RMSE per K.

We can see that between the values of k 175 and 200 the lowest RMSE occurs. Now we represent k values and MAE using the following code:

```
kValue = np.arange(2,df4_sal.shape[1],2) # To set a range to K.

plt.figure(figsize=(10,10)) # To set figure size

# To set the plot settings
width = 1/1.5
plt.plot(kValue, Avg_MAE,'go--', linewidth=1, markersize=3)

plt.xlabel('k Values') # X axis label
plt.ylabel('RMSE per K') # Y axis label
plt.grid('on')

plt.show() # To show only the plot
print(min(Avg_RMSE))
```

The following figure is the one we get by executing the code above:

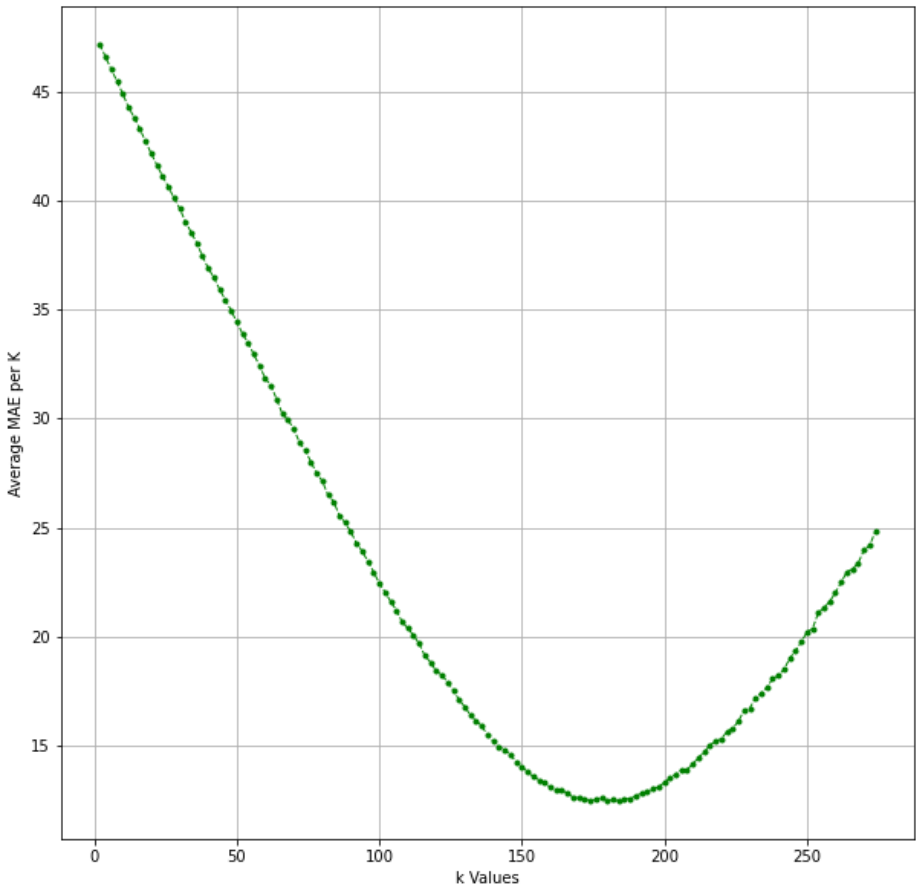

Figure 6. K values and MAE per K.

We can see that the values are repeated and between the values of k 175 and 200 the lowest MAE occurs. Now we represent k values and RMSE and MAE using the following code:

```
kValue = np.arange(2,276,2)

plt.figure(figsize=(10,10))

width = 1/1.5
plt.plot(kValue, Avg_RMSE,'go--', linewidth=1, markersize=3)
plt.plot(kValue, Avg_MAE,'go--', linewidth=1, markersize=3)

plt.xlabel('k Values')
plt.ylabel('RMSE and MAE per K')
plt.grid('on')

plt.show()
print(min(Avg_RMSE))
```

The following figure is the one we get by executing the code above:

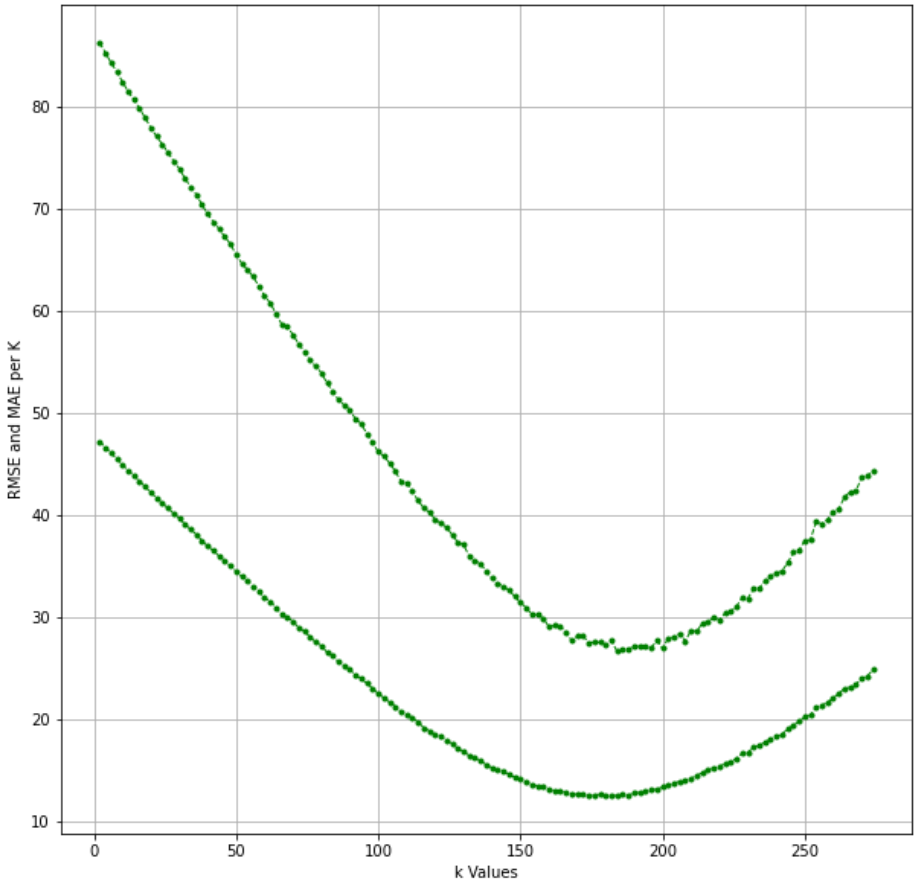

Figure 7. K values and RMSE and MAE per K.

As we can see, for the values of k between 175 and 200 we obtain the lowest values of RMSE and MAE, therefore we will establish a value of **k = 185** to train the model and obtain the predictions.

5.3. Getting recommendations

Once we have the optimal value of the model's latent factors, k=185, we can train it to infer the missing values of the user-item matrix and get recommendations. As we have already done a cross-validation process to calculate the optimal value of latent factors k, we will train the model with the entire dataset, in order to be able to infer all the missing values of the user-item matrix and to be able to make personalized recommendations for each store.

Using the following code, we train the model and get the user-item matrix approach:

```
trainset = data.build_full_trainset() # To set the trainset

# Build the model with k=185, and train it.
algo = NMF(n_factors=185, verbose=False)
algot = algo.fit(trainset)

pu = algot.pu # Get U from the model
qi = algot.qi # Get V from the model
qiT = qi.T # Transpose V to do the dot product

# Doing the dot product U.VT we get the user-item matrix approach
all_stores_predicted_sales = np.dot(pu, qiT)
```

What this code snippet returns is a numpy matrix with the approximation of the user-item matrix calculated using 185 latent factors. To transform it into a dataframe we use the following code:

```
preds_df = pd.DataFrame(all_stores_predicted_sales,
                        columns = df4_sal.columns,
                        index = df4_sal.index )
```

The resulting dataframe has the same shape as the matrix user-item shown before, but there are no missing values. Note that the known values from the original user-item values are slightly different. This is because they have been computed again using the low-rank matrices in which we decomposed the original user-item matrix before:

ITEM_DESC	A	B	C	D	E	F	G	H	I	J	L
STORE_KEY											
1701	15.26	15.82	9.72	19.88	10.61	16.94	19.62	9.27	40.25	21.51	42.35
1703	13.84	8.39	6.17	8.13	3.60	19.06	8.56	6.96	54.78	34.98	68.59
1705	10.97	6.76	6.06	6.93	3.31	10.60	6.53	10.79	84.07	23.02	52.32
1707	11.15	10.97	7.41	11.09	4.02	12.55	9.01	7.61	34.54	16.15	42.52
1709	17.18	9.89	6.29	7.54	3.75	26.65	8.25	8.50	30.62	25.45	61.55

Table 3. First 5 instances of the filled matrix user-item.

To verify that we have had no errors passing the approximation numpy matrix to a dataframe, we check a specific store and product value to see if the inferred value matches our data frame using the following code:

```
uid = str(1701)  # User id
iid = 'A'  # Item id

# get a prediction for specific users and items.
pred = algo.predict(uid, iid, r_ui=9.99, verbose=True)
```

```
user: 1701       item: A          r_ui = 9.99   est = 15.26
```

As we can see, the prediction for a specific user and item matches our dataframe value for that specific user and item, so we can be sure that the dataframe is correct.

Once we have the user-item matrix with no missing values, and having verified that the dataframe matches the numpy matrix values, we can define a **function** that returns recommendations on the products that could work best for a specific store among the products that the store does not sell.

To do this, we will first filter a specific store from the user-item array without missing values, which will give us the recommendations for the store. Then we will filter the same store of the grouped dataframe to obtain the products that the store already sells. Finally, we will remove the products that the store already sells from the recommendations and we will order the results in descending order in order to make the recommendations.

As we explained above, we first filter the recommendations by a specific store. After that, from the grouped dataframe we obtain the annualized sales by product of the specific store. We print the number of products that the store already sells, and the number of recommendations we will make. Finally, we added information to the recommendations of the class and the group of products to compare the results with those of the classification of type A stores that we saw in point 1.

This described process is implemented with the following lines of code:

```
def recommend_products(predictions_df, STORE_KEY,salest, sales, num_recommendations=5):

    # Get and sort the store's predictions
    store = str(STORE_KEY)
    sorted_store_predictions = preds_df.loc[store].sort_values(ascending=False)

    # Get the store's data
    store_data = sales[sales.STORE_KEY == store].sort_values(['SALES_WO_VAT_ML'], ascending=False)

    # Print number of products that the store already sells and number of recomendations
    print ('Store',  STORE_KEY, 'sells' , store_data.shape[0], 'products.')
    print ('Recommending highest', num_recommendations, 'predicted sold products not sold yet.')

    # Recommend the highest predicted sold products that the store hasn't sold yet.
    sortedpred = pd.DataFrame(sorted_store_predictions).reset_index()
    recommendations = (sortedpred[~sortedpred['ITEM_DESC'].isin(store_data['ITEM_DESC'])].
                       rename(columns = {store: 'Predictions'}).sort_values('Predictions',
                                                                            ascending = False).
        iloc[:num_recommendations, :])

    # We add group and class information for further comparisons
    group_class = []
    class_= []
    for x in recommendations.ITEM_DESC:
        group_class.append(salest.CWT_GROUP_CLASS_DESC[salest.ITEM_DESC == x].unique())
        class_.append(salest.CWT_CLASS_DESC[salest.ITEM_DESC == x].unique())
    recommendations['GROUP'] = group_class
    recommendations['CLASS'] = class_

    return store_data, recommendations
```

To call the function we have to specify:

- User-item matrix of recommendations.
- Specific store to which we want to make recommendations.
- General sales dataframe without grouping to extract information.
- Grouped annualized sales dataframe.
- A number of products we want to recommend.

We call the function using the following code:

```
already_sold, predictions = recommend_products(preds_df, 1701,df4, df4_sales, 10)
```

```
Store 1701 sells 158 products.
Recommending highest 10 predicted sold products not sold yet.
```

What the function returns to us are two tables, one with the products that the store already sells ordered in descending order:

```
already_sold.head(10)
```

	STORE_KEY	ITEM_DESC	SALES_WO_VAT_ML	SALES_MARKET_SHARE_STORE	UNIT_SALES
45	1701	A	223.51	0.06	169.25
29	1701	B	172.73	0.03	24.00
47	1701	C	171.73	0.05	131.08
102	1701	D	160.69	0.02	41.08
68	1701	E	151.59	0.04	277.50
37	1701	F	149.73	0.16	119.42
136	1701	G	139.82	0.04	103.83
111	1701	H	134.06	0.02	51.08
30	1701	I	118.73	0.02	14.00
70	1701	J	116.13	0.03	212.50

Table 4. First 10 instances of the products that the store already sells.

And another with as many recommendations as we have indicated to the function, with the class and group of recommended products:

	ITEM_DESC	Predictions	GROUP	CLASS
2	AR	166.65	[SNACKS]	[CHIPS]
4	BR	159.40	[SNACKS]	[CHIPS]
6	CR	143.82	[SNACKS]	[CHIPS]
8	DR	115.17	[SNACKS]	[PUFFED AND EXTRUDED SNACKS]
9	ER	113.35	[SNACKS]	[CHIPS]
11	FR	105.63	[SNACKS]	[CHIPS]
12	GR	98.01	[SNACKS]	[CHIPS]
17	HR	88.31	[SNACKS]	[CHIPS]
18	IR	86.39	[SNACKS]	[CHIPS]
22	JR	75.91	[SNACKS]	[CHIPS]

Table 5. Recommendations to a specific store.

6. Results

The results obtained in the previous section are the recommendations that the recommender system designed in this project makes to a specific store. These recommendations are intended to make type B and C stores become type A thanks to the recommendations offered by this recommender system.

PepsiCo tell us that for that chain of supermarkets and hypermarkets in that country in Europe, the group that has the most sales is SNACKS, therefore it makes sense that for a specific store, products from the best-selling group are recommended if the store does not still sell it, because the stores around it will surely get good performance with those products.

As possible points of improvement we highlight the following:

- It would be interesting to be able to personalize the recommendation for each class. So that the classes of each store that have the least sales can be reinforced.

- In this model, regularization factors of 0.02 have been considered. It would be interesting to optimize these factors to see with what value we obtain lower RMSE and MAE. The Surprise library has a functionality that optimizes all the parameters of the chosen algorithm with respect to an error metric. It would be interesting to implement this functionality.

- Our model has been trained with annualized sales. Being a wide continuous variable (0: 2452.93) the error metrics RMSE and MAE give high values compared to other matrix factorization models that use smaller ranges. Perhaps it would be interesting to discretize the sales variable in a range of 1: 5 to be able to compare the model with others.

Regarding the model, the field of matrix factorization research applied to recommender systems is extremely active. One particularly effective strategy is to combine matrix factorization and neighborhood methods into one framework, as it happens in [4].

References

[1] Xin Luo, Mengchu Zhou, Yunni Xia, and Qinsheng Zhu. (2014). An efficient non-negative matrix factorization-based approach to collaborative filtering for recommender systems. *IEEE Transactions on Industrial Informatics* Volume 10, issue:2

[2] Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. (2006). Learning from incomplete ratings using non-negative matrix factorization. *SIAM International Conference on Data Mining*, PR124.

[3] Daniel D, Lee and H. Sebastian Seung. (2001). Algorithms for non-negative matrix factorization. *Neural information processing systems foundation.* vol. 13, pp. 556–562.

[4] Yehuda Koren. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. *Association for Computing Machinery.*

Y. Koren and R. Bell. (2011). Advances in collaborative-filtering, in Recommender Systems Handbook, F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, Eds. New York, NY, USA. *Springer*, pp. 145–186.

Y. Li, B. Cao, L. Xu, J. Yin, S. Deng, Y. Yin et al. (2014). An efficient recommendation method for improving business process modeling. *IEEE Transactions on Industrial Informatics*. vol. 10, no. 1, pp. 502–513.

A. Paterek. (2007). Improving regularized singular value decomposition for collaborative-filtering. *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discov*. Data Mining, San Jose, CA, USA, pp. 39–42.

Y. Koren, R. Bell, and C. Volinsky. (2009). Matrix-factorization techniques for recommender systems. *Computer*. vol. 42, no. 8, pp. 30–37.

R. Salakhutdinov and A. Mnih. (2008). Probabilistic matrix-factorization. *Adv. Neural Inf. Process*. Syst. vol. 20, pp. 1257–1264.

G. Chen, F. Wang, and C. Zhang. (2009). Collaborative-filtering using orthogonal nonnegative matrix tri-factorization. *Inf. Process. Manage*. vol. 45, pp. 368–379.

J. Herlocker, J. Konstan, L. Terveen, and J. Riedl. (2004). Evaluating collaborative-filtering recommender systems. *ACM Trans. Inf. Syst*. vol. 22, pp. 5–53.

M. W. Berry, M. Browne, A. N. Langville,V. P. Pauca, and R. J. Plemmons. (2007). Algorithms and applications for approximate nonnegative matrix-factorization. *Comput. Statist. Data Anal*. vol. 52, pp. 155–173.